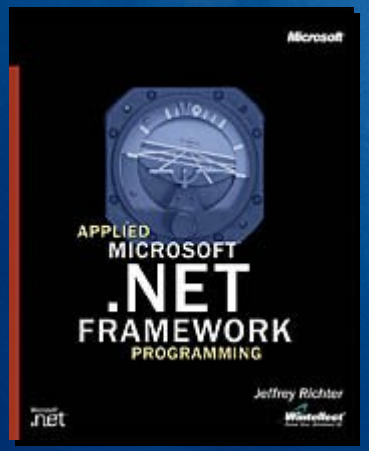
DEV490

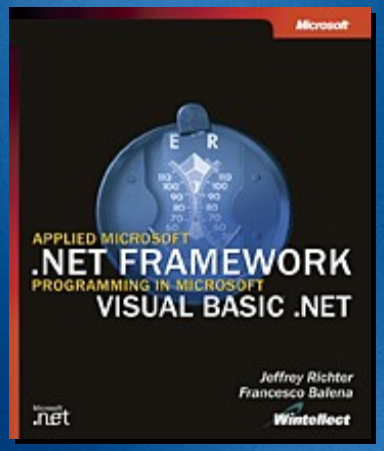
.NET Framework: CLR – Under The Hood

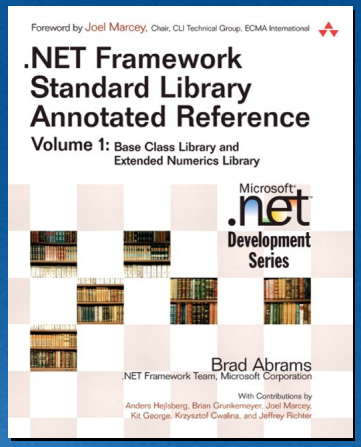
Jeffrey Richter
Author / Consultant / Trainer
Wintellect

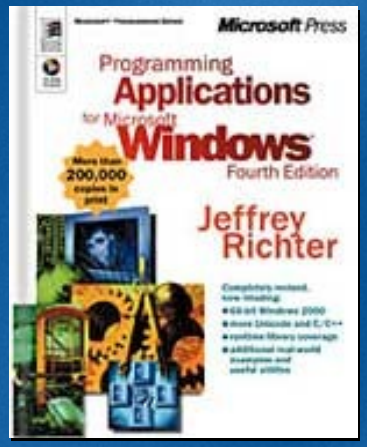
Jeffrey Richter

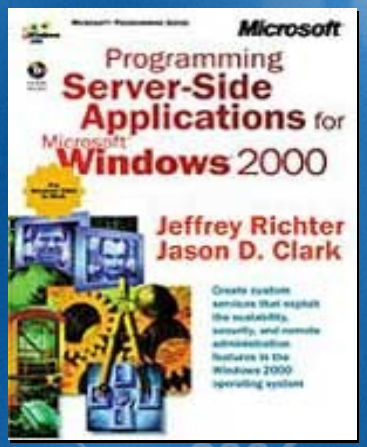
- Author of several .NET Framework/Win32 Books
- Cofounder of Wintellect: a company dedicated to helping clients ship better software faster
  - Services: Consulting, Debugging, Security Reviews, Training
- Consultant on Microsoft's .NET Framework team since October 1999
- MSDN Magazine Contributing Editor/.NET Columnist











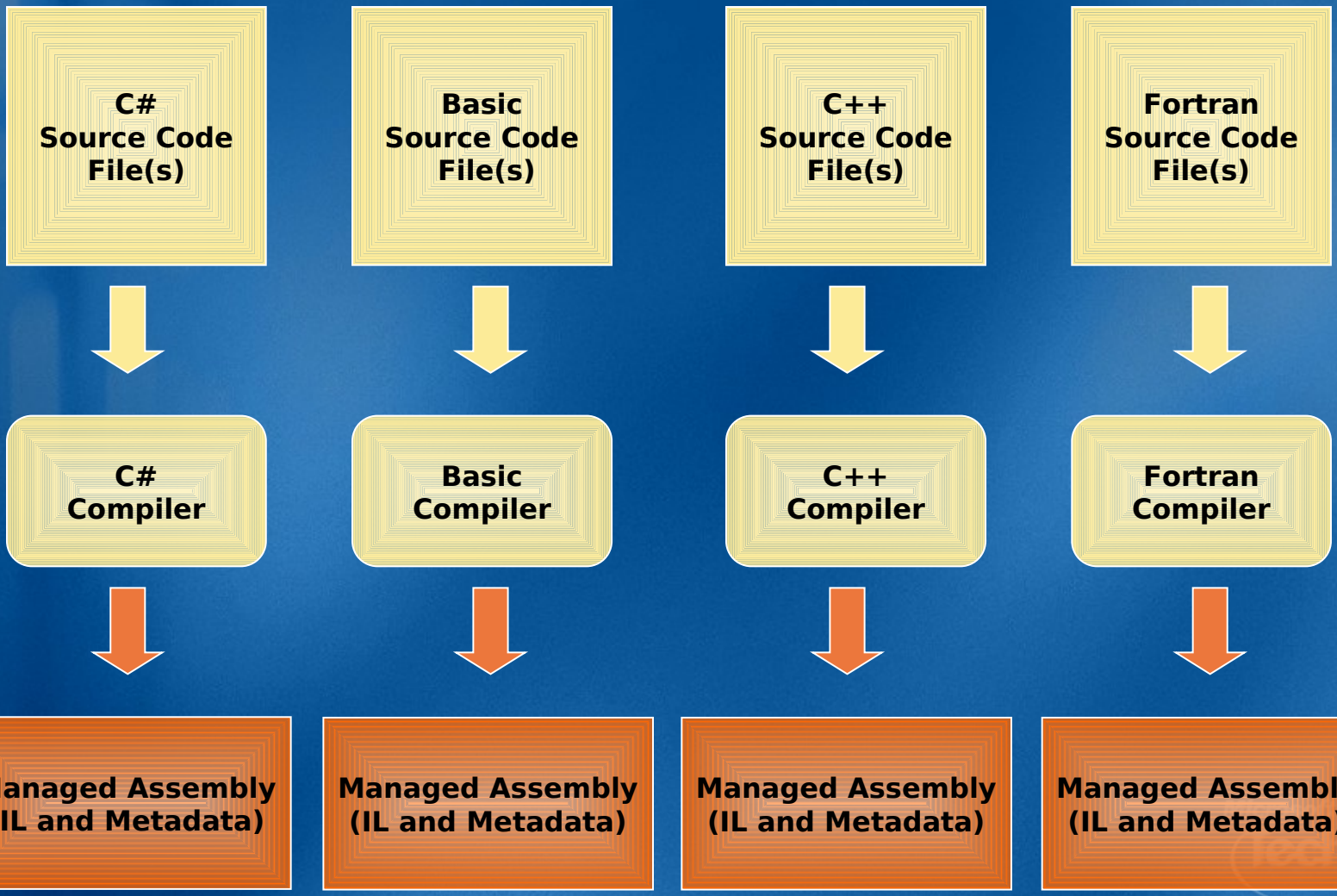
Topics

- Execution Model
  - Intermediate Language (IL), verification, JIT compilation, metadata, and assembly loading
- How Things Relate at Runtime
  - Code, Types, Objects, a thread's stack, and the heap
- Garbage Collection
  - How a reference-tracking GC works

Topics

- Execution Model
  - Intermediate Language (IL), verification, JIT compilation, metadata, and assembly loading
- How Things Relate at Runtime
  - Code, Types, Objects, a thread's stack, and the heap
- Garbage Collection
  - How a reference-tracking GC works

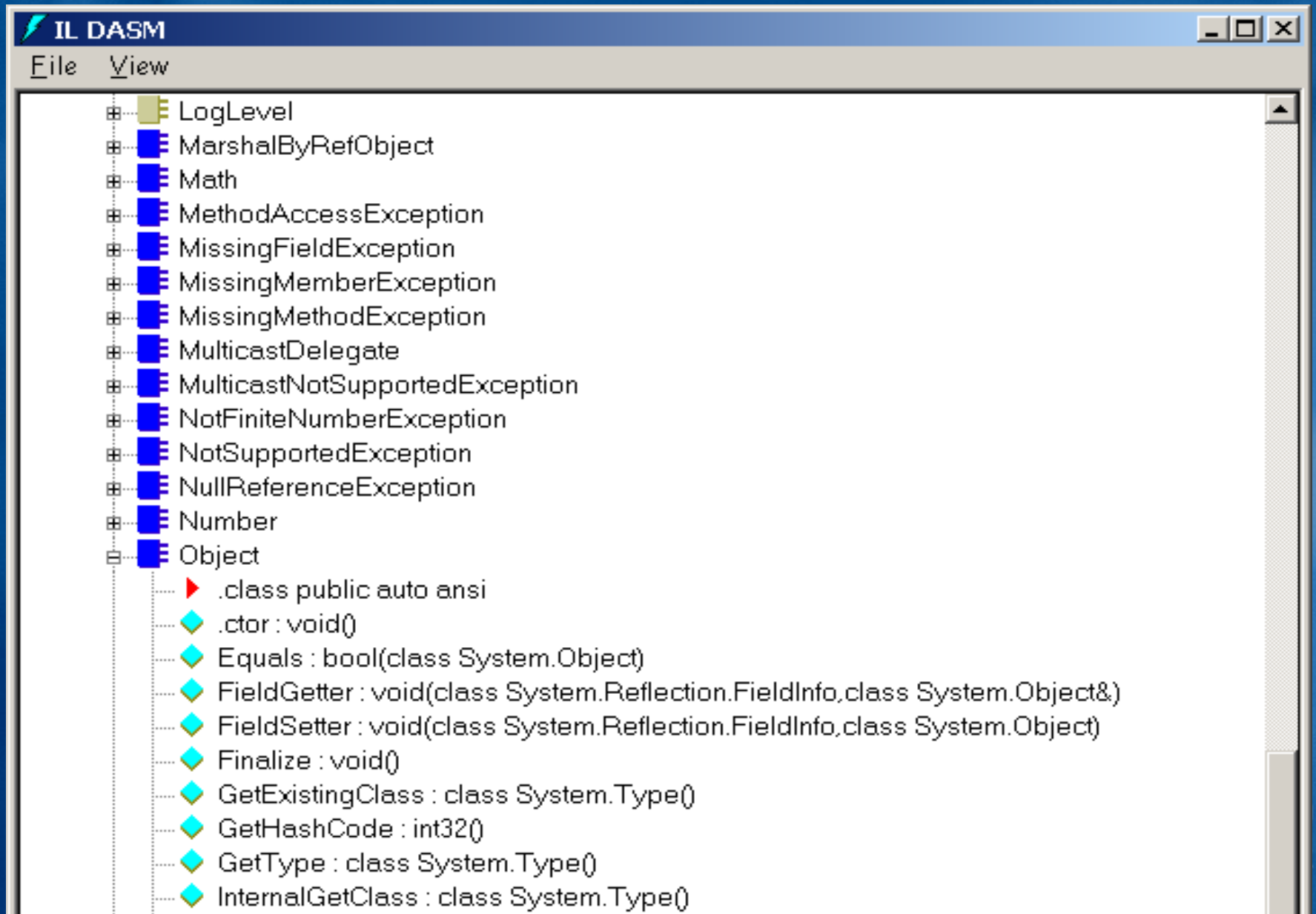
Compiling Source Code Into Assemblies



An Assembly

- An Assembly is the managed equivalent of an EXE/DLL
- Implements and optionally exports a collection of types
- It is the unit of versioning, security, and deployment
- Parts of an Assembly file
  - Windows PE header
  - CLR header (Information interpreted by the CLR and utilities)
  - Metadata (Type definition and reference tables)
  - Intermediate Language (code emitted by compiler)

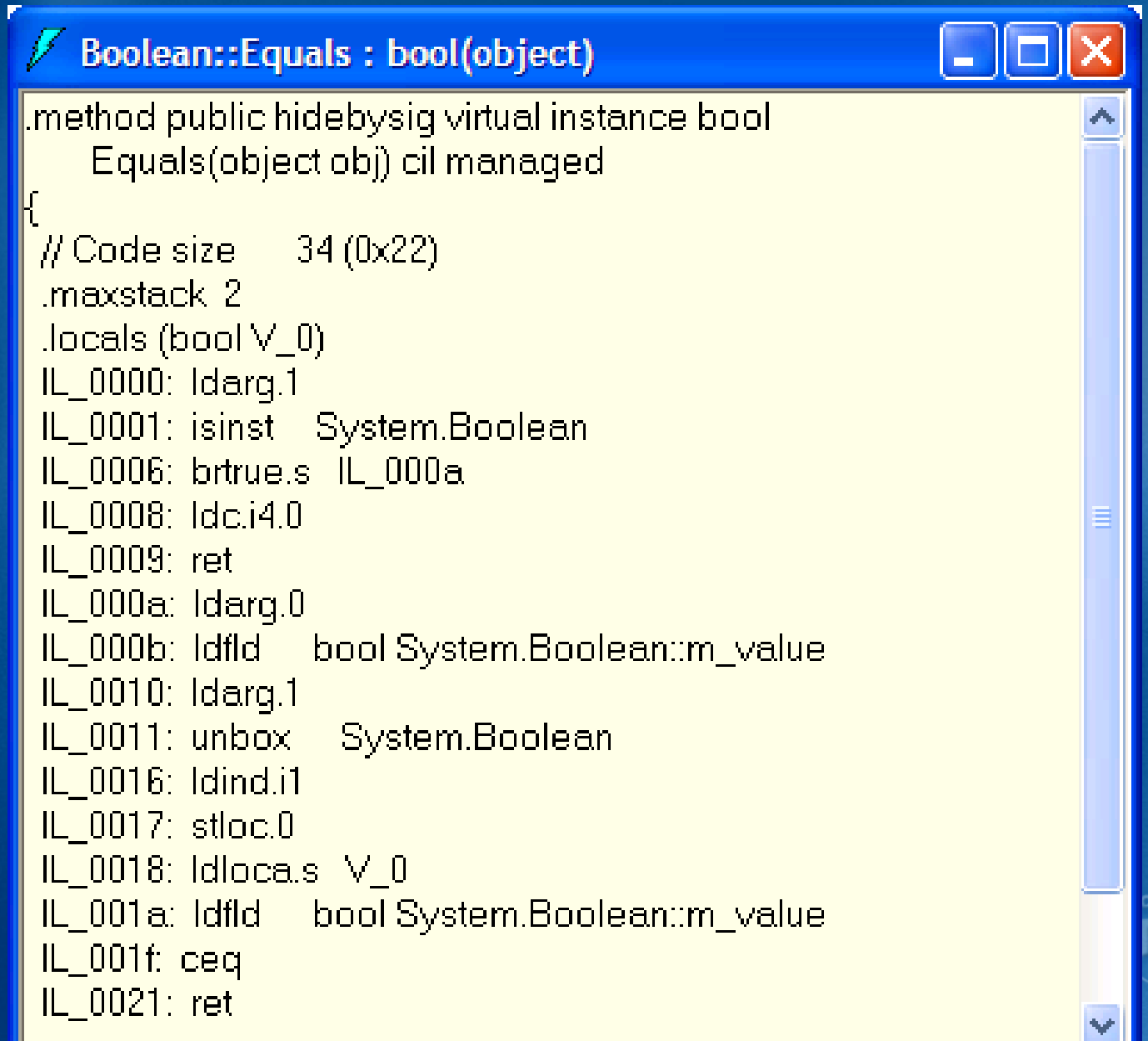
ILDasm.exe


IL DASM
File View
LogLevel
MarshalByRefObject
Math
MethodAccessException
MissingFieldException
MissingMemberException
MissingMethodException
MulticastDelegate
MulticastNotSupportedException
NotFiniteNumberException
NotSupportedException
NullReferenceException
Number
Object
.class public auto ansi
.ctor : void()
Equals : bool(class System.Object)
FieldGetter : void(class System.Reflection.FieldInfo,class System.Object&)
FieldSetter : void(class System.Reflection.FieldInfo,class System.Object)
Finalize : void()
GetExistingClass : class System.Type()
GetHashCode : int32()
GetType : class System.Type()
InternalGetClass : class System.Type()

Intermediate Language

- All .NET compilers produce IL code
- IL is CPU-independent machine language
  - Created by Microsoft with input from external commercial and academic language/compiler writers
- IL is higher-level than most CPU machine languages
- Some sample IL instructions
  - Create and initialize objects (including arrays)
  - Call virtual methods
  - Throw and catch exceptions
  - Store/load values to/from fields, parameters, and local variables
- Developers can write in IL assembler (ILAsm.exe)
  - Many compilers produce IL source code and compile

ILDasm.exe


Boolean::Equals : bool(object)
.method public hidebysig virtual instance bool
Equals(object obj) cil managed
{
// Code size 34 (0x22)
.maxstack 2
.locals (bool V_0)
IL_0000: ldarg.1
IL_0001: isinst System.Boolean
IL_0006: brtrue.s IL_000a
IL_0008: ldc.i4.0
IL_0009: ret
IL_000a: ldarg.0
IL_000b: ldfld bool System.Boolean::m_value
IL_0010: ldarg.1
IL_0011: unbox System.Boolean
IL_0016: ldind.i1
IL_0017: stloc.0
IL_0018: ldloca.s V_0
IL_001a: ldfld bool System.Boolean::m_value
IL_001f: ceq
IL_0021: ret

Benefits Of IL

- IL is not tied to any specific CPU
- Managed modules can run on any CPU (x86, Itanium, Opteron, etc), as long as the OS on that CPU supports the CLR
- Many believe that “write once, run everywhere” is the biggest benefit
  - *I disagree, security and verification of code is the really BIG win!*

Real Benefit Of IL: Security And Verification

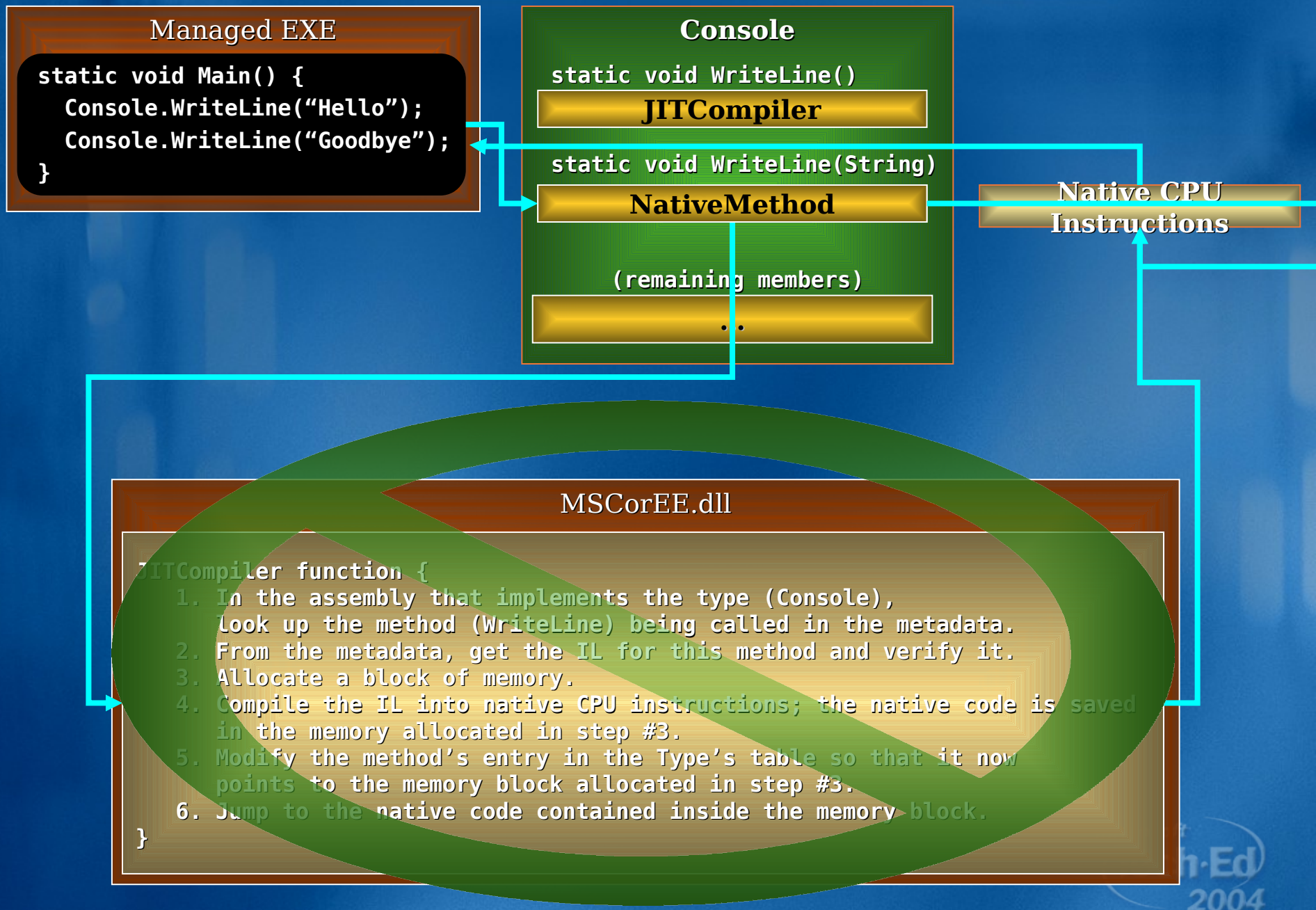
- When processing IL, CLR verifies it to ensure that everything it does is “safe”
  - Every method is called with correct number and type of parameters
  - Every method’s return value is used properly
  - Every method has a return statement
- Metadata includes all the type/method info used for

Benefits Of "Safe" Code

- Multiple managed applications can run in 1 Windows' process
  - Applications can't corrupt each other (or themselves)
- Reduces OS resource usage, improves performance
- Administrators can trust apps
  - ISPs forbidding ISAPI DLLs
  - SQL Server running IL for stored procedures
  - Internet downloaded code (with Code Access Security)
- Note: Administrator can turn off verification

Executing Managed IL Code

- When loaded, the runtime creates method stubs
- When a method is called, the stub jumps to runtime
- Runtime loads IL and compiles it
  - IL is compiled into native CPU code
  - Just like compiler back-end
- Method stub is removed and points to compiled code
- Compiled code is executed
- In future, when method is called, it just runs


Managed EXE
static void Main() {
Console.WriteLine("Hello");
Console.WriteLine("Goodbye");
}
Console
static void WriteLine()
JITCompiler
static void WriteLine(String)
NativeMethod
(remaining members)
..
Native CPU Instructions
MSCorEE.dll
JITCompiler function {
1. In the assembly that implements the type (Console),
look up the method (WriteLine) being called in the metadata.
2. From the metadata, get the IL for this method and verify it.
3. Allocate a block of memory.
4. Compile the IL into native CPU instructions; the native code is saved
in the memory allocated in step #3.
5. Modify the method's entry in the Type's table so that it now
points to the memory block allocated in step #3.
6. Jump to the native code contained inside the memory block.
}

All Types/Modules Are Self-Describing

```
public class App {
   public static void Main() {
      System.Console.WriteLine("Hi");
   }
}
```

- 1 TypeDef entry for “App”
  - Entry refers to MethodDef entry for “Main”
- 2 TypeRef entries for “System.Object” and “System.Console”
  - Both entries refer to AssemblyRef entry for MSCorLib

Metadata Definition Tables (Partial List)

- TypeDef: 1 entry for each type defined
  - Type's name, base type, flags (i.e. public, private, etc.) and index into MethodDef & FieldDef tables
- MethodDef: 1 entry for each method defined
  - Method's name, flags (private, public, virtual, static, etc), IL offset, and index to ParamDef table
- FieldDef: 1 entry for each field defined

Metadata Reference Tables (Partial List)

- AssemblyRef: 1 entry for each assembly ref'd
  - Name, version, culture, public key token
- TypeRef: 1 entry for each type ref'd
  - Type's name, and index into AssemblyRef table
- MemberRef: 1 entry for each member ref'd
  - Name, signature, and index into TypeRef table

demo

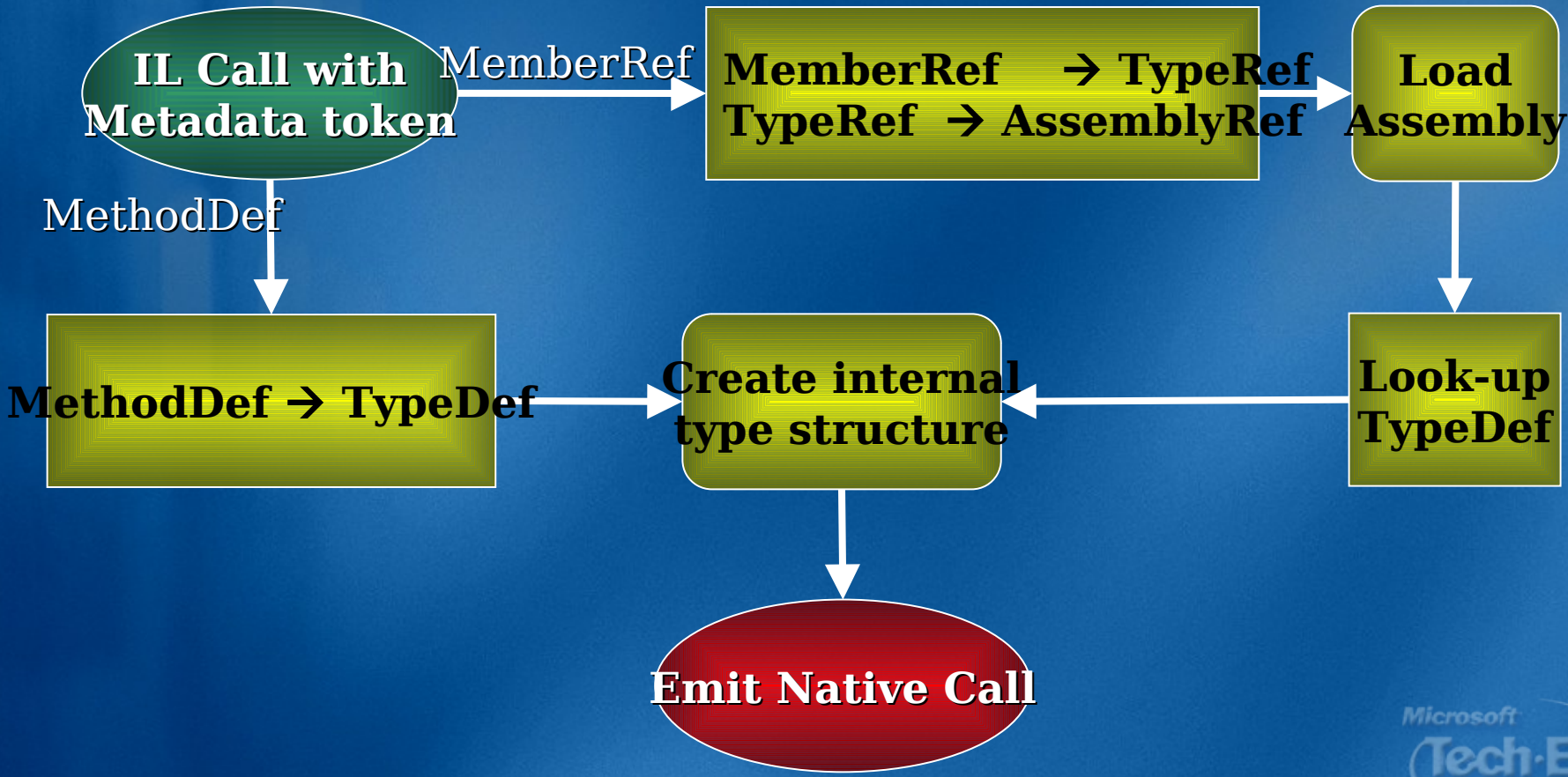
ILDasm.exe Metadata And IL

Code Attempts To Access A Type/Method

```
.method /*06000001*/ public hidebysig static
        void  Main(class System.String[] args) il managed
{
  .entrypoint
  // Code size       11 (0xb)
  .maxstack  8
  IL_0000:  ldstr      "Hi“
  IL_0005:  call       void ['mscorlib'/* 23000001 */]
     System.Console/* 01000003 */::WriteLine(class System.String)
  IL_000a:  ret
} // end of method 'App::Main'
```

- 23000001: AssemblyRef entry for MSCorLib
- 01000003: TypeRef entry to System.Console

How The CLR Resolves An Assembly Reference



Topics

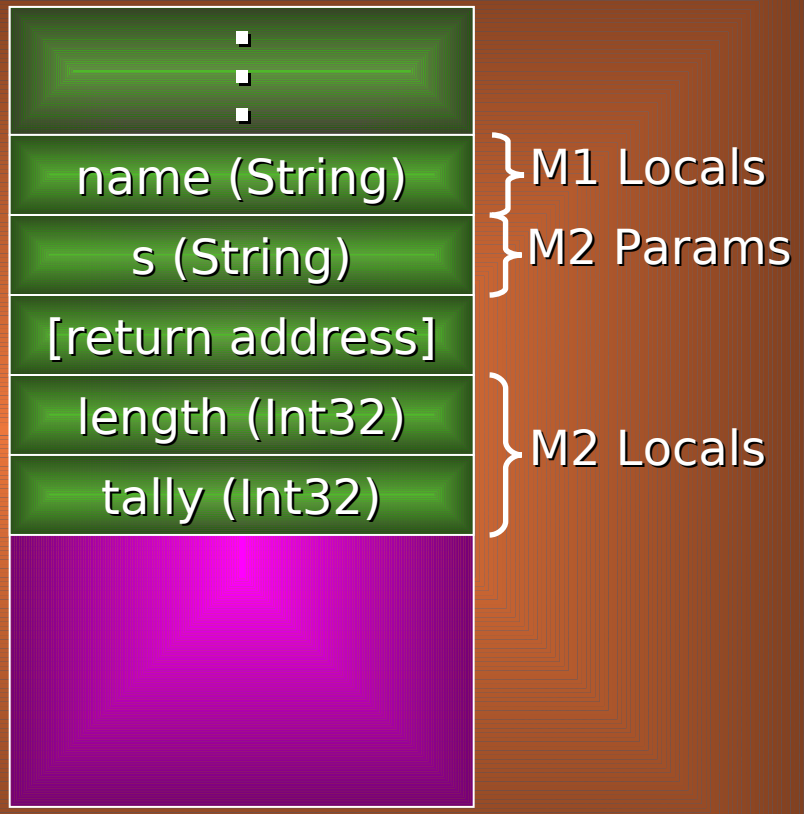
- Execution Model
  - Intermediate Language (IL), verification, JIT compilation, metadata, and assembly loading
- How Things Relate at Runtime
  - Code, Types, Objects, a thread's stack, and the heap
- Garbage Collection
  - How a reference-tracking GC works

A Thread's Stack

Windows' Process

```
void M1() {
  String name = "Joe";
  M2(name);
  ...
  return;
}
```

```
void M2(String s) {
  Int32 length = s.Length;
  Int32 tally;
  ...
  return;
}
```



CLR (Thread Pool & Managed Heap)

Simple Class Hierarchy

```
class Employee {
   public         Int32     GetYearsEmployed()   { ... }
   public virtual String    GenProgressReport()  { ... }
   public static  Employee  Lookup(String name)  { ... }
}
```

```
class Manager : Employee {
   public override String   GenProgressReport()  { ... }
}
```

Instance Method Mapping Using *this*

```
public Int32 GetYearsEmployed();
  ↳ public (static) Int32 GetYearsEmployed(Employee this);

public virtual String GenProgressReport();
  ↳ public (static) String GenProgressReport(Employee this);

public static  Employee  Lookup(String name);
  ↳ public static  Employee  Lookup(String name);
```

```
Employee e = new Employee();
e.GetYearsEmployed();
e.GenProgressReport();
```

```
Employee e = new Employee();
Employee.GetYearsEmployed(e);
Employee.GenProgressReport(e);
```

- ***this*** **is what makes instance data available to instance methods**

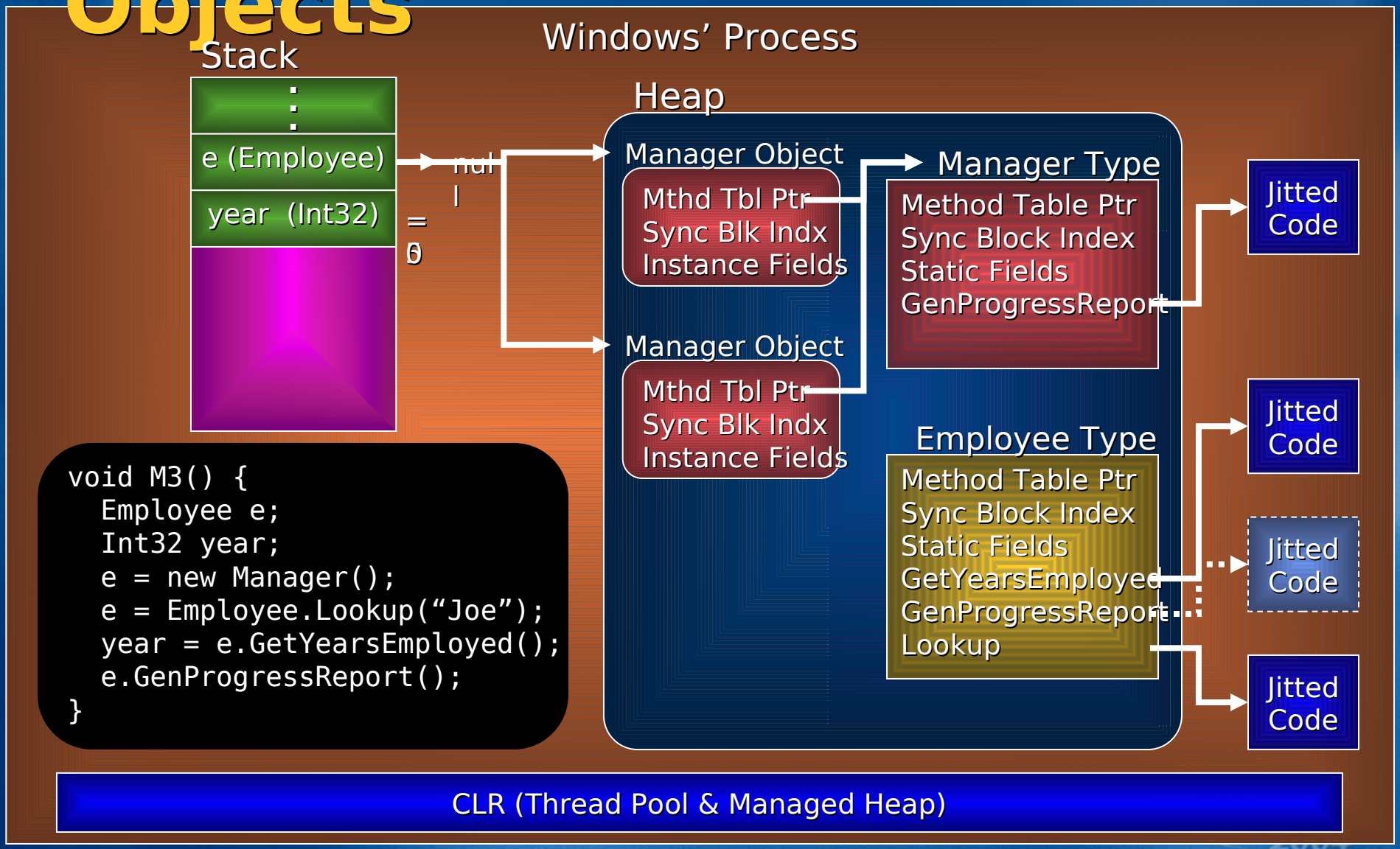
IL Instructions To Call A Method

- Call
  - Is usable for static, instance, and virtual instance methods
  - No null check for the *this* pointer (for instance methods)
  - Used to call virtual methods non-polymorphically
    - base.OnPaint();
- Callvirt
  - Usable for instance and virtual methods only
  - Slower perf
  - Null check for all instance methods
  - Polymorphs for virtual methods
    - No polymorphic behavior for non-virtual methods
  - C# and VB use *callvirt* to perform a null check when

```
class App {
   static void Main() {
      Object o = new Object();
      o.GetHashCode();       // Virtual
      o.GetType();           // Non-virtual instance
      Console.WriteLine(1);  // Static
   }
}
```

```
.method private hidebysig static void  Main() cil managed {
  .entrypoint
  // Code size       27 (0x1b)
  .maxstack  1
  .locals init (object V_0)
  IL_0000:  newobj     instance void System.Object::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  callvirt   instance int32 System.Object::GetHashCode()
  IL_000c:  pop
  IL_000d:  ldloc.0
  IL_000e:  callvirt   instance class System.Type System.Object::GetType()
  IL_0013:  pop
  IL_0014:  ldc.i4.1
  IL_0015:  call       void System.Console::WriteLine(int32)
  IL_001a:  ret
} // end of method App::Main
```

Memory: Code, Types, Objects

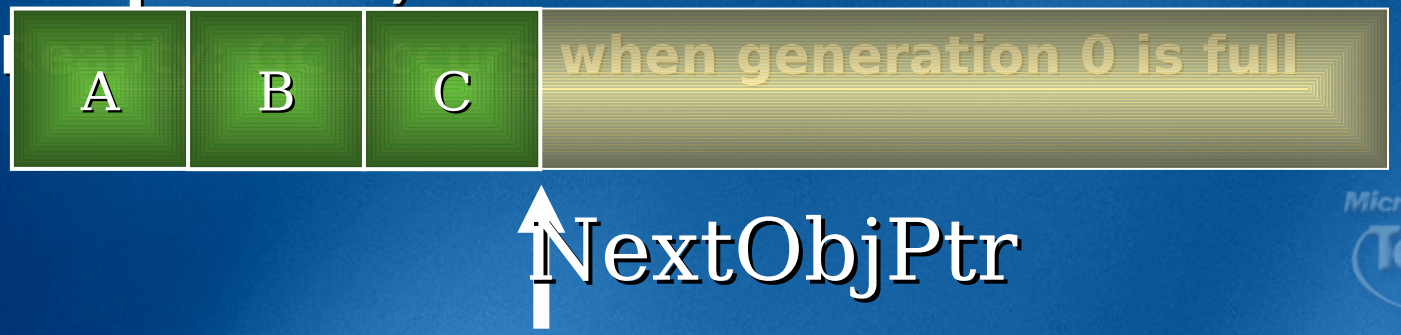


Topics

- Execution Model
  - Intermediate Language (IL), verification, JIT compilation, metadata, and assembly loading
- How Things Relate at Runtime
  - Code, Types, Objects, a thread's stack, and the heap
- Garbage Collection
  - How a reference-tracking GC works

The Managed Heap

- All reference types are allocated on the managed heap
  - Your code never frees an object
  - The GC frees objects when they are no longer reachable
- Each process gets its own managed heap
  - Virtual address space region, sparsely allocated
- The new operator always allocates objects at the end
- If heap is full, a GC occurs
  - [illegible] when generation 0 is full



Roots And GC Preparation

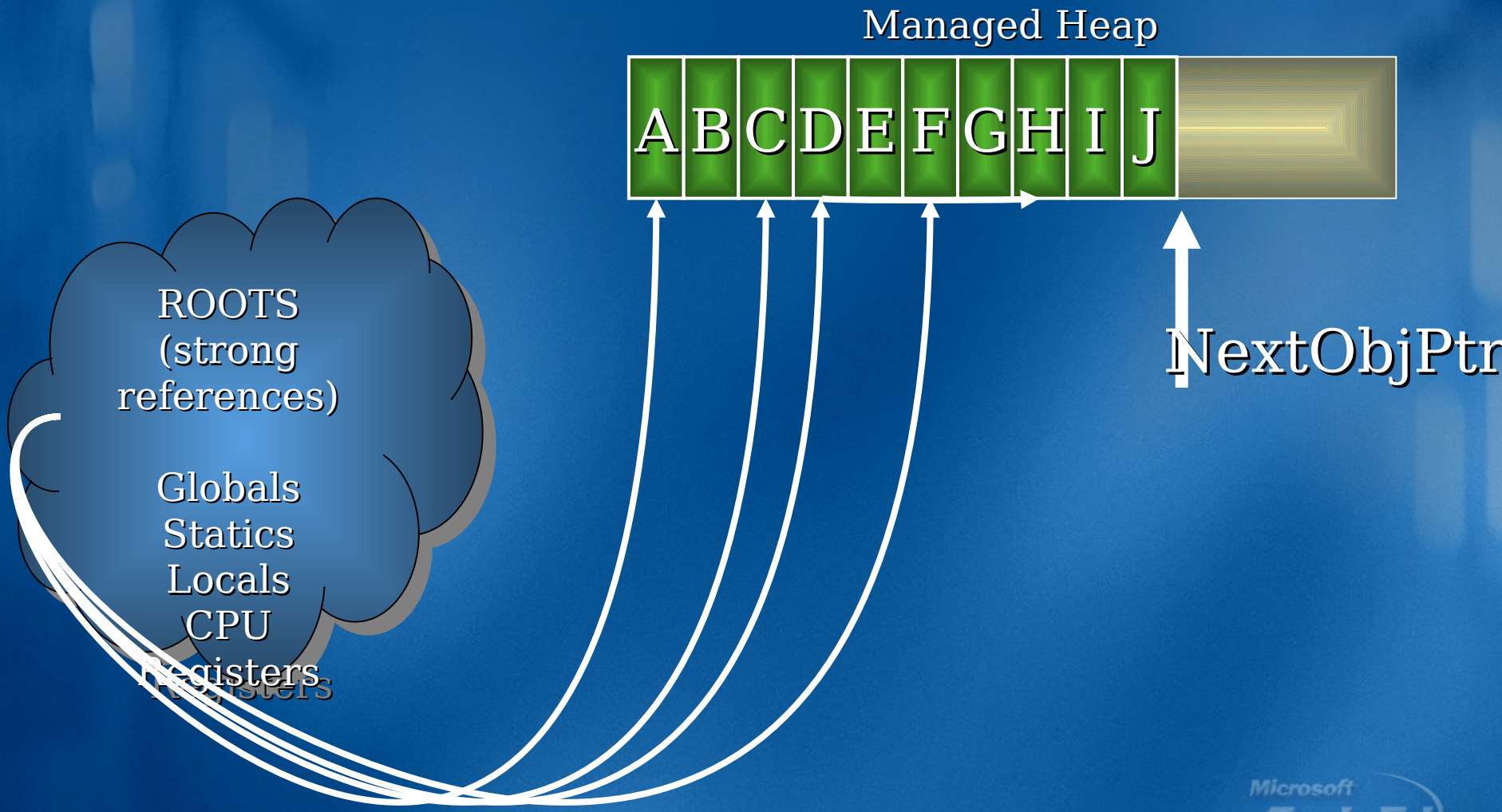
- Every application has a set of Roots
- A Root is a memory location that can refer to an object
  - Or, the memory location can contain null
- Roots can be any of the following
  - Global & static fields, local parameters, local variables, CPU registers
- When a method is JIT compiled, the JIT compiler creates a table indicating the method's roots
  - The GC uses this table
  - The table looks something like this...

Start Offset	End Offset	Roots
0x00000000	0x00000020	this, arg1, arg2, ECX, EDX
0x00000021	0x00000122	this, arg2, fs, EBX
0x00000123	0x00000145	fs

When A GC Starts…

- All objects in heap are considered garbage
  - The GC assumes that no roots refer to objects
- GC examines roots and marks each reachable object
  - If a GC starts and the CPU's IP is at 0x00000100, the objects pointed to by the this parameter, arg2 parameter, fs local variable, and the EBX register are roots; these objects are marked as "in use"
    - As reachable objects are found, GC uses metadata to checks each object's fields for references to other objects
      - These objects are marked "in use" too, and so on
  - GC walks up the thread's call stack determining roots for the calling methods by accessing each method's table
    - For objects already "in use", fields aren't checked
      - Improves performance
      - Prevents infinite loops due to circular references
- The GC uses other means to obtain the set of roots stored in global and static variables

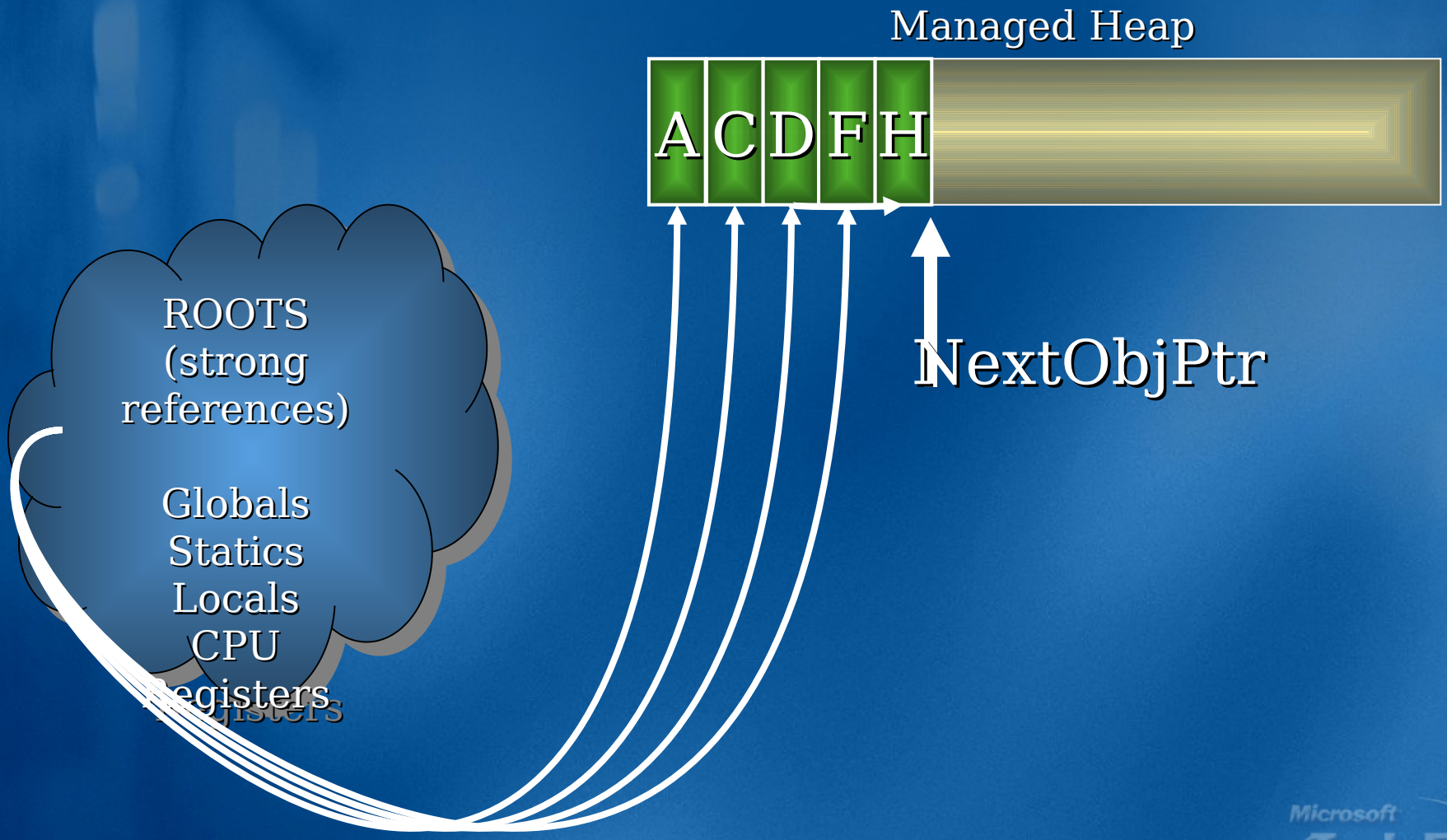
Before A Collection



Compacting The Heap

- After all roots have been checked and all objects have been marked “in use”…
  - The GC walks linearly through heap for free gaps and shifts reachable objects down (simple memory copy)
  - As objects are shifted down in memory, roots are updated to point to the object’s new memory address
- After all objects have been shifted…
  - The NextObjPtr pointer is positioned after last object
  - The ‘new’ operation that caused the GC is retried
    - This time, there should be available memory in the heap and the object construction should succeed
      - If not, an OutOfMemoryException is thrown

After A Collection

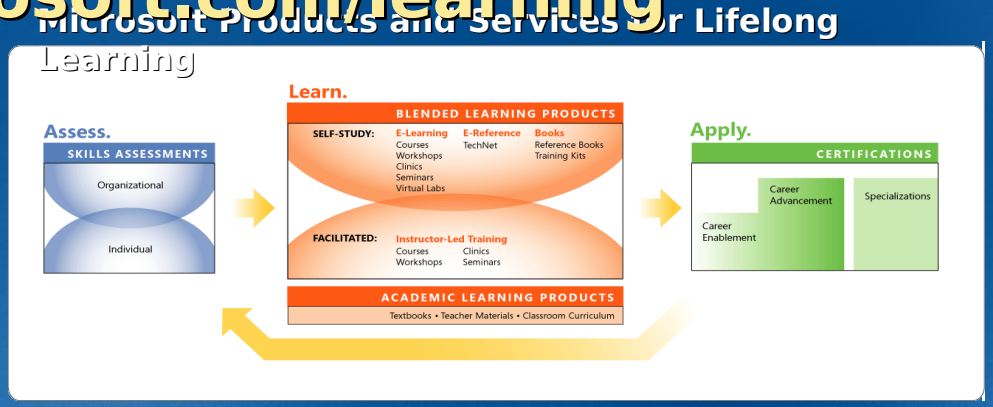


Root Example

```
class App {
   public static void Main() {
      // ArrayList object created in heap, a is now a root
      ArrayList a = new ArrayList();
      // Create 10000 objects in the heap
      for (int x = 0; x < 10000; x++) {
         a.Add(new Object());
      }

      // Local a is a root that refers to 10000 objects
      Console.WriteLine(a.Length);
      // After line above, a is not a root and all 10001
      // objects may be collected.
      // NOTE: Method doesn't have to return
      Console.WriteLine("End of method");
   }
}
```

Microsoft Products And Services For Lifelong Learning
www.microsoft.com/learning



Assessments	• http://www.microsoft.com/assessment/
Courses	• 2415: Programming with the Microsoft .NET Framework (Microsoft Visual Basic .NET) • 2349: Programming with the Microsoft .NET Framework (Microsoft Visual C# .NET)
Books	• The Microsoft Platform Ahead, ISBN: 0-7356-2064-4 • Network Programming for the Microsoft .NET Framework, ISBN: 0-7356-1959-X • Programming Microsoft .NET, ISBN: 0-7356-1376-1 • Applied Microsoft Windows .NET Framework Programming, ISBN: 0-7356-1422-9 • Applied Microsoft Windows .NET Framework Programming in Microsoft Visual Basic .NET, ISBN: 0-7356-1787-2 • Microsoft Windows .NET Framework 1.1 Class Library References • Performance Testing Microsoft .NET Web Applications, ISBN: 0

DEV490

.NET Framework: CLR – Under The Hood

Jeffrey Richter
Author / Consultant / Trainer
Wintellect